# Introduction to Algorithms

Mong-Jen Kao (高孟駿)

Tuesday 10:10 – 12:00

Thursday 15:30 – 16:20

# Data Structures

Particular ways of storing data *to support special operations*.

# Hash Tables & Hash Functions

A data structure that ***performs extremely well in practice*** for the *dictionary* operations.

Also a data structure that allows us to *escape from* ***the natural barriers*** of ***comparison-based algorithms***.

# Natural Barrier of Comparison-Based Algorithms

- We have seen that, ***comparison-based algorithms*** for the ***sorting*** problem requires $\Omega(n \log n)$ time to solve.
  - Achieved by *Quicksort*, *Heapsort*, *Mergesort*, etc.
- We have also seen that, *with further prior-knowledge* given for the input, sorting in $O(n)$ time is possible.
  - For example, *counting sort*, *radix sort*, *bucket sort*, etc.
  - In essence, all of these algorithm achieves the $O(n)$ running time by ***mapping the input elements properly***.

This is what a ***hashing function*** does.

# Natural Barrier of Comparison-Based Algorithms

- We have seen (in the midterm exam problems) that,
  it takes $\boldsymbol{\Omega(n \log n)}$**-time** for any comparison-based algorithm to solve
  ***the element uniqueness*** (EU) problem.

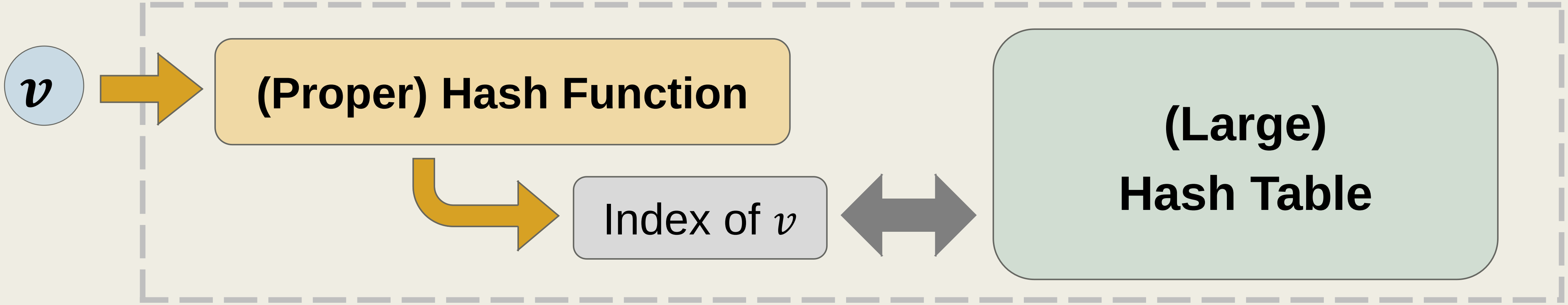
- We will see in this lecture that, *with proper assumptions*,
  the EU problem can be solved in expected $O(n)$ time.
  - Many problems can be solved *more efficiently and easily*
    if there is a proper way to **map the elements** to a ***certain domain***.

# Hash Tables

A data structure that ***performs extremely well in practice*** for the *dictionary* operations.

# Hash Table

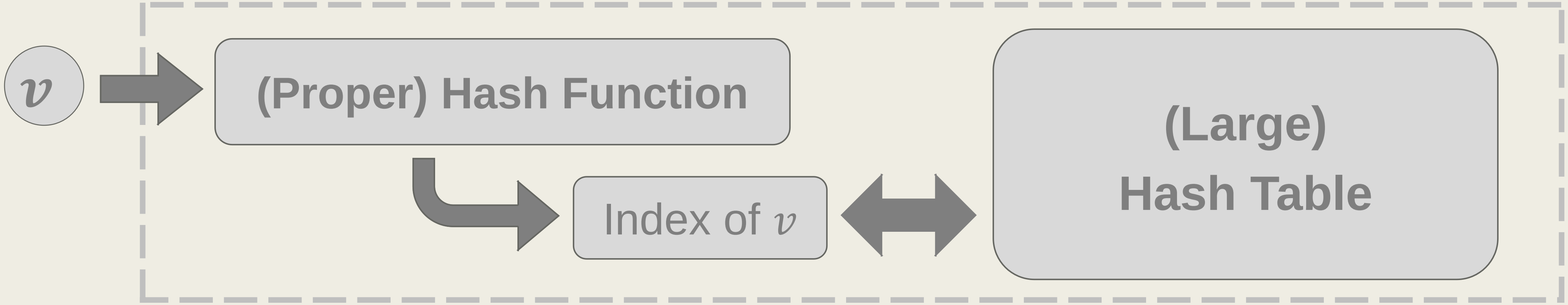
- In general, ***hash table*** is a data structure that supports the *dictionary operations* such as ***Insert***, ***Search***, and ***Delete***.
  - Under reasonable assumptions,
    these operations take $O(1)$ time in average. (!)
- To process a given element $v$, we use a (proper) hash function to compute ***the supposed index of*** $v$ **in the hash table**.



- To process a given element $v$, we use a (proper) hash function to compute ***the supposed index of*** $v$ **in the hash table**.
  - Let $m$ be the number of slots in the hash table.
  - A ***hash function***

$$h : U \mapsto \{ 0, 1, \ldots, m - 1 \}$$

    maps the ***universe*** $U$ ***of all possible keys*** to *the slots in the table*.
  - Then, *insertion*, *search*, and *deletion* are done ***accordingly***.



# Independent Uniform Hash Functions

- An ideal hash function $h$ would have the property that
  - For each key $k$ in the domain $U$,
    the output $h(k)$ is an element chosen ***(uniformly) randomly***
    and ***independently*** from $\{0, 1, \ldots, m-1\}$.

- We call such an ideal hash function an *independent uniform hash function*.
  - Such a function is also referred to as a ***random oracle***.

The result of hashing appears to be uniformly random.

- An ideal hash function $h$ would have the property that
  - For each key $k$ in the domain $U$,
    the output $h(k)$ is an element chosen ***(uniformly) randomly*** and ***independently*** from $\{ 0, 1, \ldots, m-1 \}$.
- We call such an ideal hash function an independent uniform hash function.
  - Such a function is also referred to as a ***random oracle***.
- The result (without prior knowledge) appears to be random.
  - After the first call, any ***subsequent call*** returns ***the same result***.

# Density / Load Factor of the Elements

- Let $T$ be a hash table with $m$ slots that stores a total number of $n$ elements.
  - We define the load factor of $T$ to be $\alpha \coloneqq n/m$.
- With *independent uniform hashing*, the ***expected number of elements*** stored ***in each slot*** would be $\alpha$.

# Resolving the Collisions

- When *multiple elements* are *mapped to the same index* by the hash function we use, we have a ***collision***.

- There are two different ways to handle collisions.

  1. Store the elements ***in place*** with another data structure.
     - Store the elements with a *linked list* (*chain*).
     - Use a *second hash table*.
     - Use a BST, etc.

$O(1+\alpha)$ time ***in average***,
$O(n)$ in the ***worst-case***.

**Effective**
in practice.

??

- When *multiple elements* are *mapped to the same index* by the hash function we use, we have a ***collision***.

- There are fundamentally two different ways to handle collisions.

1. Store the elements ***in place***.

2. ***Open addressing***.

   - Store ***at most one element in each slot***.
   - Upon collision,
     ***store*** the element ***in the next slot available***.
     (search till the next empty slot)

We will discuss this approach later.

# Hash Functions

# Hash Functions

- Recall that, we prefer ideal hash functions that provide ***independent uniform hashing*** guarantees.

- If a *fixed, static* hash function is used, then...
  - The performance will be ***determined by the distribution*** of the input data set.
  - If *the adversary knows the hash function*, he/she can choose ***a set of keys*** that would be ***hashed to the same slot***.
  - Then the time it takes *for each operation* becomes $\boldsymbol{\omega}(\mathbf{1})$.

# Random Hash Functions

- Recall that, we prefer ideal hash functions that provide ***independent uniform hashing*** guarantees.
- To achieve the goal, one solution is to choose a hash function ***randomly*** *from* ***a set of hash functions*** with good properties.
  - This is the concept of ***universal hashing***.

# Uniform Family of Hash Functions

- Let $\boldsymbol{\mathcal{U}}$ be the universe of *<u>all possible keys</u>*, and $\boldsymbol{\mathcal{H}}$ be a ***<u>family of hash functions</u>*** that maps $\mathcal{U}$ into the range $\{0,1,2,\dots,m-1\}$.
  - $\mathcal{H}$ is ***<u>uniform</u>*** if

$$\Pr_{h\leftarrow\mathcal{H},\, k\leftarrow\mathcal{U},\, q\leftarrow\{0,\dots,m-1\}}[\, h(k) = q \,] \;=\; \frac{1}{m}\,.$$

  - i.e., when $h$ is picked ***<u>uniformly at random</u>*** from $\mathcal{H}$, then, for every $k \in \mathcal{U}$ and every slot $q \in \{0,1,\dots,m-1\}$, the probability that $k$ is hashed to $q$ is equal to $1/m$.

Every slot is *<u>equally likely</u>*.

# Universal Family of Hash Functions

- Let $\mathcal{H}$ be a ***family of hash functions*** mapping $\mathcal{U}$ into $\{0,1,2,\ldots,m-1\}$.
  - $\mathcal{H}$ is ***universal*** if

$$\Pr_{h\leftarrow\mathcal{H},\,k_1,k_2\leftarrow\mathcal{U}}[\,h(k_1)=h(k_2)\,]\ \le\ \frac{1}{m}\,.$$

  - i.e., when $h$ is picked ***uniformly at random*** from $\mathcal{H}$, then,
    for every $k_1, k_2 \in \mathcal{U}$,
    the probability that $k_1$ and $k_2$ result in a ***collision*** is **at most $1/m$**.

Note that, $1/m$ is the best possible when $|\mathcal{U}| \ge m$.

# $\epsilon$-Universal Family of Hash Functions

- Let $\mathcal{H}$ be a ***family of hash functions*** mapping $\mathcal{U}$ into $\{0,1,2,\ldots,m-1\}$.
  - $\mathcal{H}$ is $\epsilon$***-universal*** if

$$\Pr_{h \leftarrow \mathcal{H},\, k_1,\ldots,k_d \leftarrow \mathcal{U}}[\, h(k_1) = h(k_2)\,] \ \leq \ \epsilon\,.$$

  - i.e., when $h$ is picked ***uniformly at random*** from $\mathcal{H}$, then,

    for every $k_1, k_2 \in \mathcal{U}$,

    the probability that $k_1$ and $k_2$ result in a ***collision*** is **at most** $\epsilon$.

Here $\epsilon \geq 1/m$ (as a relaxed notion) when $|\mathcal{U}| \geq m$.

# $d$-Independent Family of Hash Functions

- Let $\mathcal{H}$ be a ***family of hash functions*** mapping $\mathcal{U}$ into $\{0,1,2,\ldots,m-1\}$.
  - $\mathcal{H}$ is $d$***-independent*** if

$$\Pr_{h \leftarrow \mathcal{H},\, \mathbf{k_1},\mathbf{k_2} \leftarrow \mathcal{U},\, \mathbf{q_1},\ldots,\mathbf{q_d} \leftarrow \{\mathbf{0},\ldots,\mathbf{m-1}\}}[\; h(k_i) = q_i \;\; \forall 1 \le i \le d \;] \le \frac{1}{m^d} \,.$$

  - i.e., when $h$ is picked ***uniformly at random*** from $\mathcal{H}$, then,

    for every subset $K \subseteq \mathcal{U}$ of keys with $|K| \le d$,

    $h$ hashes the keys in $K$ ***independently***.

# Ideal Hash Functions

- Recall that, we prefer ideal hash functions that provide ***independent uniform hashing*** guarantees.

- Let $\mathcal{U}$ be the universe of *all possible keys*, and $\mathcal{H}$ be a ***family of hash functions*** that maps $\mathcal{U}$ into the range $\{0,1,2,\ldots,m-1\}$.

- ***Independent uniform hashing*** can be achieved if we have a family of hash functions that is ***uniform***, ***universal***, and $|\mathcal{U}|$-***independent***.
  - In the following,
    we discuss some practical constructions.

(Perhaps) too good to be true in practice.

# Universal Hashing

# Universal Family of Hash Functions

- We describe a (uniform) ***universal*** family of hash functions with a ***certain degree*** of ***independence***.
  - Let $\mathcal{U}$ be the universe of keys that are (short) ***nonnegative integers***.
- Let $p$ be a sufficiently large prime number such that $\mathcal{U} \subseteq [0, p-1]$.
  - Then, $\mathbb{Z}_p = \{0,1, \dots, p-1\}$ is a field with
    - Multiplicative group $\mathbb{Z}_p^* = \{1, \dots, p-1\}$ and
    - Additive group $\mathbb{Z}_p = \{0,1, \dots, p-1\}$.

# Designing a Universal Family of Hash Functions

- Let $\mathcal{U}$ be the universe of keys that are nonnegative integers.
  - Let $p$ be a prime number such that $\mathcal{U} \subseteq [0, p-1]$.

- For any $a \in \mathbb{Z}_p^*$ and $b \in \mathbb{Z}_p$, define

$$h_{\{a,b\}}(k) \coloneqq (\ (ak+b) \bmod p\ ) \bmod m\ ,$$

where $k \in \mathcal{U}$ is the key to be hashed and $m$ is the number of slots.

# Designing a Universal Family of Hash Functions

- For any $a \in \mathbb{Z}_p^*$ and $b \in \mathbb{Z}_p$, define

$$h_{\{a,b\}}(k) \coloneqq \left( (ak + b) \bmod p \right) \bmod m \,,$$

where $k$ is the key to be hashed.

**Theorem 1.**

The family $H_{p,m} \coloneqq \left\{ h_{\{a,b\}} \,:\, a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p \right\}$ is uniform, ***universal***, and ***2-independent***.

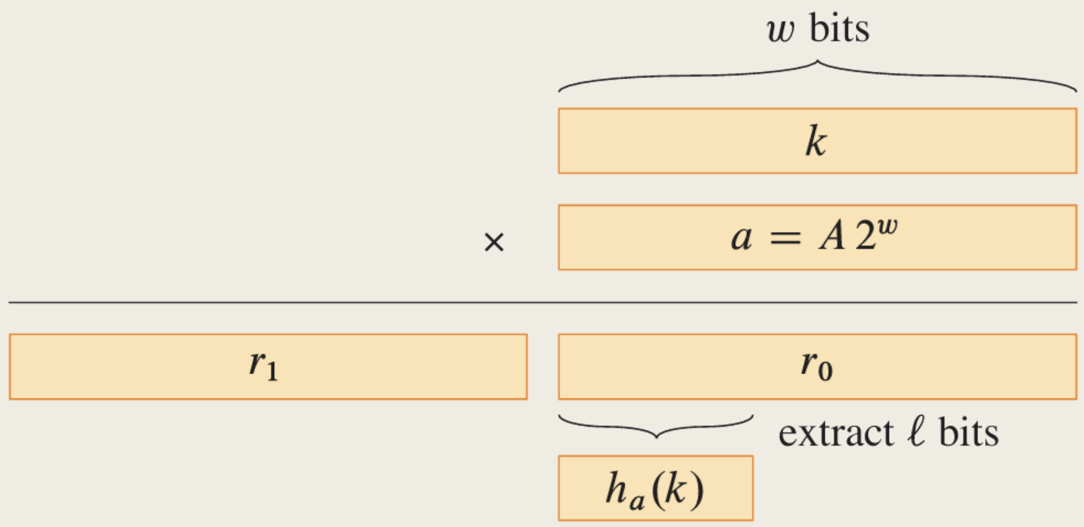
# Another Practical Construction

- Suppose that the keys are $w$-bit integers.

- Let $0 < a < 2^w$ and $0 \leq \ell \leq w$ be two chosen parameters.

  Define

$$h_a(k) \coloneqq (\ (k \cdot a \mod 2^w\ ) \gg (w - \ell)\ ).$$



# Another Practical Construction

- Suppose that the keys are $w$-bit integers.

- Let $0 < a < 2^w$ and $0 \leq \ell \leq w$ be two chosen parameters.

  Define

$$h_a(k) \coloneqq (\ (k \cdot a \mod 2^w\ ) \gg (w - \ell)\ ).$$

**Theorem 2.**

The family $H \coloneqq \{\ h_a : 1 \leq a < m,\ a \text{ odd}\ \}$ is $(2/m)$***-universal***.

# Hashing Long Inputs

# Hashing Long Inputs

- We have seen how hashing can be done for keys that are (short) non-negative integers.
- For long inputs, such as ***vectors*** or ***strings***, one can ***convert*** the input *into short non-negative integers*.
- Possible approaches includes
  - Number-theoretic Theory
  - Cryptographic Hashing

# Open-Addressing

# Resolving Collisions via Open-Addressing

- In the open-addressing scheme, we consider ***hash functions*** of the following form

$$h' : \mathcal{U} \times \{0,1,\dots,m-1\} \longrightarrow \{0,1,\dots,m-1\}$$

such that $\{ h'(k,i) \}_{0 \le i < m}$ is a ***permutation*** of $\{0,1,\dots,m-1\}$.

- We will store ***at most one element in each slot.***

- To process an operation, we ***consider*** $h'(k,i)$ for $i = 0,1,\dots,m-1$ ***in order*** until the desired operation is done.

# Resolving Collisions via Open-Addressing

- We will consider $h'(k, i)$ for $i = 0,1, \dots, m-1$ in order until the desired operation is done.
  - For insertion, we find the smallest $i$ such that $h'(k, i)$ is *“empty”* or *“deleted”* and insert $k$ at $h'(k, i)$.
  - For search, we iterate over $i$ until either the key $k$ is found, $h'(k, i)$ is *“empty”*, or $i = m - 1$.
  - ***For deletion***, however, we have to mark the entry as ***“deleted”*** instead of *“empty”*.

# Resolving Collisions via Open-Addressing

- Intuitively, when collision happens,
  we "probe" the slots in a certain order (permutation).

- There are basically two different ways to probe the slots.
  - ***Linear probing*** – to test the slots starting from $h(k)$ in order,
    i.e., $h'(k,i) \coloneqq h(k) + i \mod m$.
  - Double hashing – to use a second hash function for probing,
    i.e., $h'(k,i) \coloneqq (h_1(k) + h_2(i)) \mod m$.